AFRL-IF-WP-TR-2002-1525

TECHNOLOGY-INDEPENDENT, MULTICOMPONENT SYNTHESIS ENVIRONMENT

Dr. Moon-Jung Chung

Michigan State University
Department of Computer Science
A709 Wells Hall
East Lansing, MI 48824-1027

DECEMBER 2001

Final Report for 30 March 1992 – 09 February 1996

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

Darrell Barker

DARRELL BARKER
Project Engineer
Embedded Info Sys Engineering Branch
Information Technology Division

James S. Williamson

JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* December 2001	**2. REPORT TYPE** Final	**3. DATES COVERED *(From - To)*** 03/30/1992 – 02/09/1996

4. TITLE AND SUBTITLE TECHNOLOGY-INDEPENDENT, MULTICOMPONENT SYNTHESIS ENVIRONMENT	**5a. CONTRACT NUMBER** F33615-92-C-1029
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62204F
6. AUTHOR(S) Dr. Moon-Jung Chung	**5d. PROJECT NUMBER** 6096
	5e. TASK NUMBER 20
	5f. WORK UNIT NUMBER 23
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Michigan State University Department of Computer Science A709 Wells Hall East Lansing, MI 48824-1027	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB, OH 45433-7334	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/IFTA
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2002-1525

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

As the complexity of the microelectronics systems and the tools involved increases, design process becomes the key issue in improving productivity. In this project, we report a novel scheme of managing design process to increase the productivity and improve the quality of design. The framework provides the following facilities: 1) coordination of activities, 2) modular approach to customizing and reconfiguring processes, 3) sharing of data and processes, and 4) reuse of data and processes. The framework is based on process grammar, a formal representation of design process. The strong theoretical foundation of our approach allows users to analyze and predict the behavior of a particular process. The abstraction of process is separated from the execution environment, proving a natural way of browsing process repository, and allowing process reuse and improvement. The execution environment determines an individual organization's requirements, mapping them to the most suitable tools and process. Individuals may, if necessary, modify the flow to match an organization's particular needs. The execution environment coordinates the activities (invoking tools, workflow, etc.) with the correct data at the right time. The framework guides the designer toward a design that meets diverse criteria such as performance, cost, design time, and safety.

15. SUBJECT TERMS

process grammar, design process management, workflow management, electronic computer-aided design

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Darrell Barker
Unclassified	Unclassified	Unclassified	SAR	56	**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-6548 x3605

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

List of Figures

1. Introduction

The increasing complexity of system design and the emergence of new technologies make the design process the key issue in the microelectronics and computer-aided design (CAD) industries [1]. A complex system design has the following characteristics: hierarchical design, multiple design representations, and a large design space. A large system design typically includes multiple boards, a variety of implementation technologies, and interfaces. Design engineers must deal with many issues, such as partitioning among different boards, interface timing, and packaging.

There are three important issues in system design. The first issue is design modeling.

Design modeling defines the functionalities of the design and verifies its correctness by simulation. These models not only ensure design correctness but also greatly affect synthesis results. Design, methodology or workflow management in a Computer Aided Design (CAD) framework must support a seamless way of carrying out the design process as well as a suitable way of representing the design process. It must also support tool encapsulation in order to carry out the design process. Design data management deals with storing design data and capturing relationships such as version control and configuration binding. Design modeling and methodology management in relation to a novel execution environment and framework developed by this author (see description below) comprise the main issues discussed in this thesis. For a detailed analysis of design data management, see Kim [2].

CAD frameworks are design environments consisting of design tools that aid design activities. The CAD framework's support for the design process has three parts: specification, execution, and services. Specification corresponds to how tasks can be

decomposed, what tools are available, and how they may be used. Execution is concerned with what methodology or process to select for a given task, what tool to invoke, and how to invoke it. Services support the coordination of subprocesses and enforce consistency. Although the concept of the CAD framework can be applied to many different engineering design disciplines, the discussion here is concerned with high-level synthesis only.

2. Methodology Management

Methodology is a set of processes or approaches used to solve a given problem. Methodology management is a technique employed to control these processes or approaches so that a better solution can be found. In the CAD area, design methodology management provides “the definition, presentation, execution, and control of design methodologies in a flexible, configurable way” [3]. The goals of design methodology management are to help the designer reduce the design time and to produce ~ better design.

In the last couple of decades, there has been a change in trends in the CAD community. The main focus of the CAD framework has shifted from managing data and tools to managing the design process itself.

Design methodology management should provide specification methods, an execution environment, and miscellaneous services. To choose appropriate specification methods, the following questions must be answered: How can tasks be decomposed? What tools are available? How will they be used? In the execution environment, the management system must provide a means of selecting the appropriate design methodology or process for a given task and must determine the choice and method of tool invocation. Miscellaneous services, such as graphical user interface for communicating between the user and system, supporting cooperating subprocesses in the system, and enforcing consistent designs, should also be included in the execution environment.

The basic building blocks of a design methodology management system or a CAD framework are tools. In general, a tool cannot be decomposed into any subcomponents; thus, the CAD framework has no way to break a tool down into smaller tools. Each tool performs a specific function. A design methodology management system determines how to use these tools as well as when to use them. The sequence of tool usage is viewed as a design flow.

A large electronic design has the following characteristics:

- Hierarchical Design: A large design can itself be hierarchically organized, and the same is true for the design process. The whole design process can be broken down into several steps or subprocesses, where each subprocess can again be decomposed into multiple sub-subprocesses. A large system design can be partitioned into smaller functional subcomponents. Each of these subcomponents can be composed of other components. Multiple design teams can work together to produce this design.

- Multiple Design Representations: A design process can be viewed as a series of data transformations from one representation to another. Each transformation produces a different. type of design data. Furthermore, the same design transformation can be used with different sets of constraints and design parameters. Consequently, each of these transformations creates a different version of the data.

- Large Design Space: Many alternative processes for a task create many different versions of data, just as different values of design parameters lead to different results. As the size of design data increases, the time required to search the design database in- creases as well. Thus, a large design space should be maintained such that an efficient way of searching through the database is possible.

- Large Number of Tools: Many tools are involved in the system design process. These tools should be well maintained so that the right tool is used at the right time.

These characteristics lead to the use of an integrated design environment or CAD framework that can manage such issues effectively and guide designers to produce viable designs,

In [4], a CAD framework is defined as "a software infrastructure that provides a common operating environment for CAD tools," In order for such a software infrastructure to provide a good integrated design environment, the CAD framework, should support the following services: Design data management, Design methodology management, Tool integration/encapsulation, and User interface [5]:

- Design data management*:* Design data management deals with the methods used to store and retrieve design data as well as maintain relationships (such as version, trans- formation, and configuration) and consistencies between designs. In the CAD design process, many different data files (either different intermediate results or different versions from the same task) must be stored and retrieved as needed, Thus, an efficient way of managing design data is necessary. This also includes version and configuration management. Design data management assists in the use of technology-independent design data. If the data is technology independent, these data can be reused in different design processes without changing them.

- Design Methodology Management: This should provide a formal representation method of the design process and a seamless way of carrying out the design process. Design methodology management selects the best tool for a given input and constraints, and guides the designer to produce the best design, In other words, design methodology management is responsible for selecting and executing an appropriate sequence of tools to produce a desired design adhering to the given specifications, Thus, it guides the user in selecting the right tools in the correct order. The CAD framework should also support concurrent engineering concepts. For these reasons, design methodology management has gained a lot of attention in the past couple of decades.

- Tool Integration: CAD tools can be integrated into the design environment by using a well-defined tool integration method. In order to handle many different tools which accept different types of input and produce different types of output, the CAD framework must provide an intertool communication mechanism. This facility ensures that all tools in the system can communicate with each other.

- User Interface: The user interface should be easy to use and effective. It should also hide low-level implementation details as much as possible from the end user (e.g., tool-invoking sequences and commands).

The block diagram of such a CAD framework is shown in Figure 1. The CAD framework helps the designer to reduce design time and errors to produce a better solution,

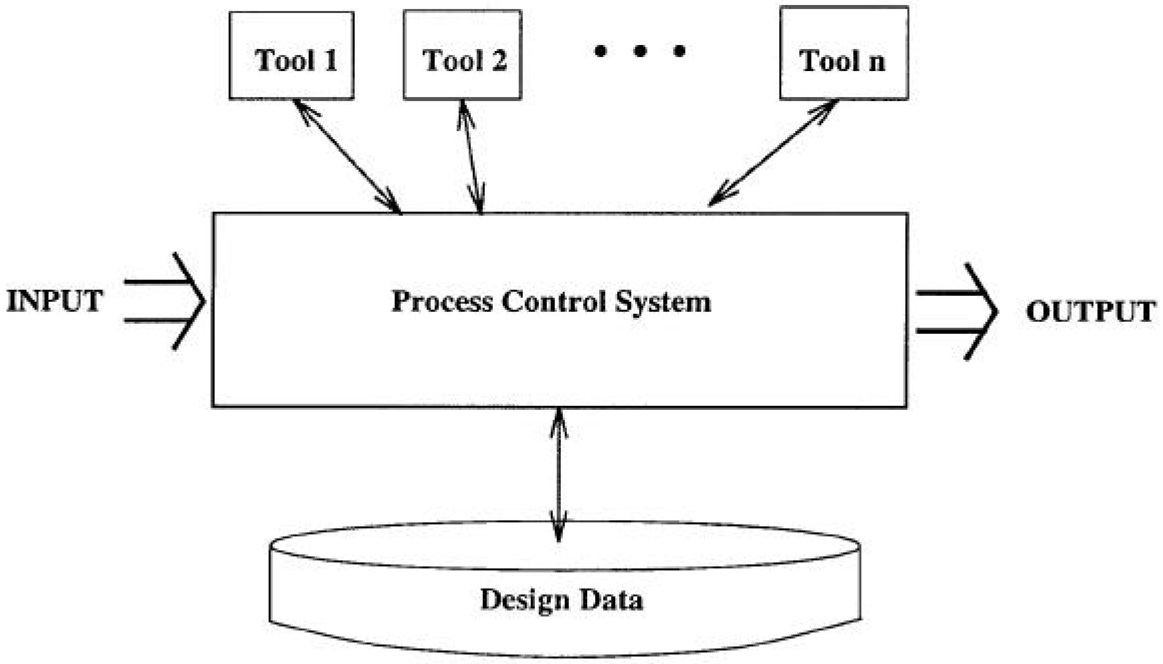


Figure 1. Black Diagram of CAD Framework

2.1 Specification Hierarchy

Task specifications are defined and organized into a specialization and generalization class hierarchy. Properties of general task specifications are also available in a special task specification where the general task specification is a parent of the special task specification. A child specification inherits its parent's specification properties. For example, pre- and post-evaluation conditions can be inherited by children; however, any child specification can have its own condition through specialization.

It is possible to decompose tasks hierarchically into simpler tasks until the individual tasks can be performed by single tool invocations. Methodologies are devised by hierarchically decomposing logical tasks until all tasks are terminal. A single tool selection can be considered to be a special case of decomposition in which the set of subtasks is a single terminal task. A specification node definition editor window is shown in Figure 2. An example of task hierarchy, the Layout Synthesis Task is shown in Figure 3. Among many layout design styles, three common layout design styles are Gate Array, Standard Cell, and Full Custom. These design styles can be applied to any level of layout synthesis hierarchy. These style conditions can be passed on to children tasks in the hierarchy. However , Multichip Module (MCM) Layout should have different conditions to be imposed than the conditions for Printed Wiring Board (PCB) Layout, and Chip Layout because in MCM layout, interchip delay can be ignored.

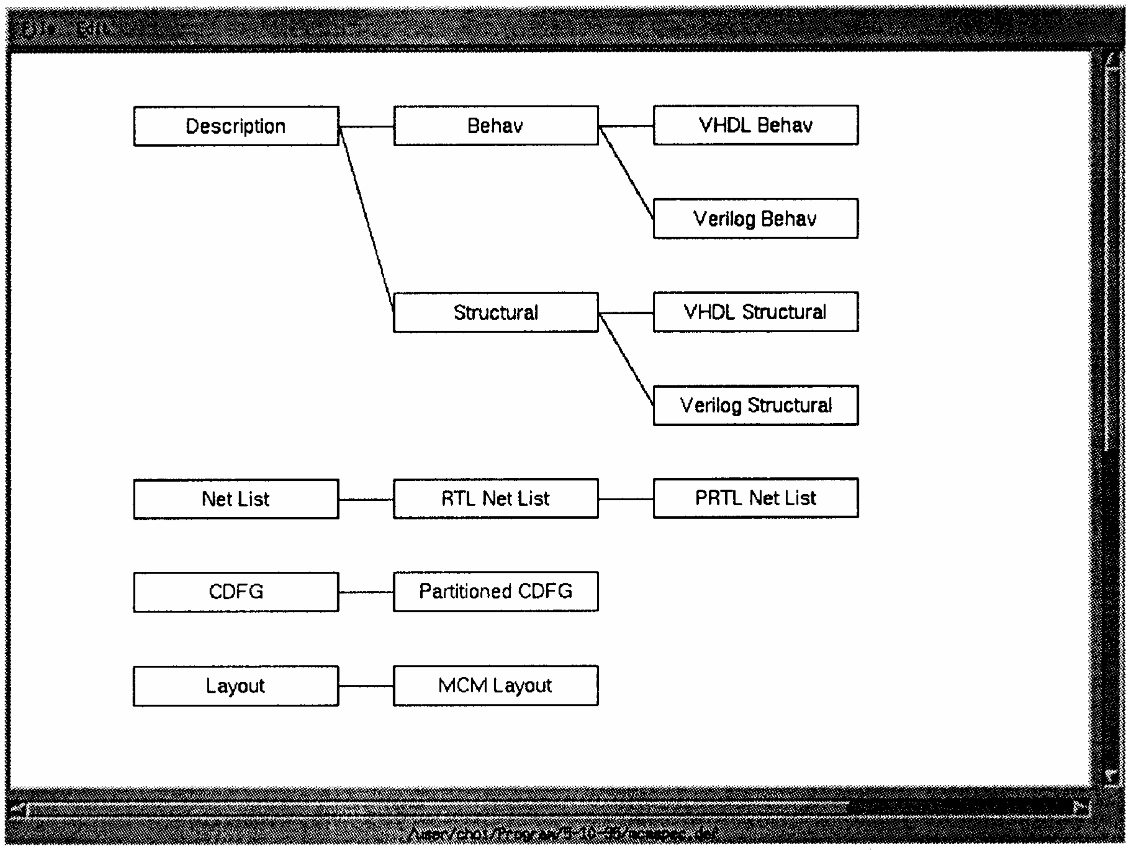


Figure 2. Specification Definition Editor Window

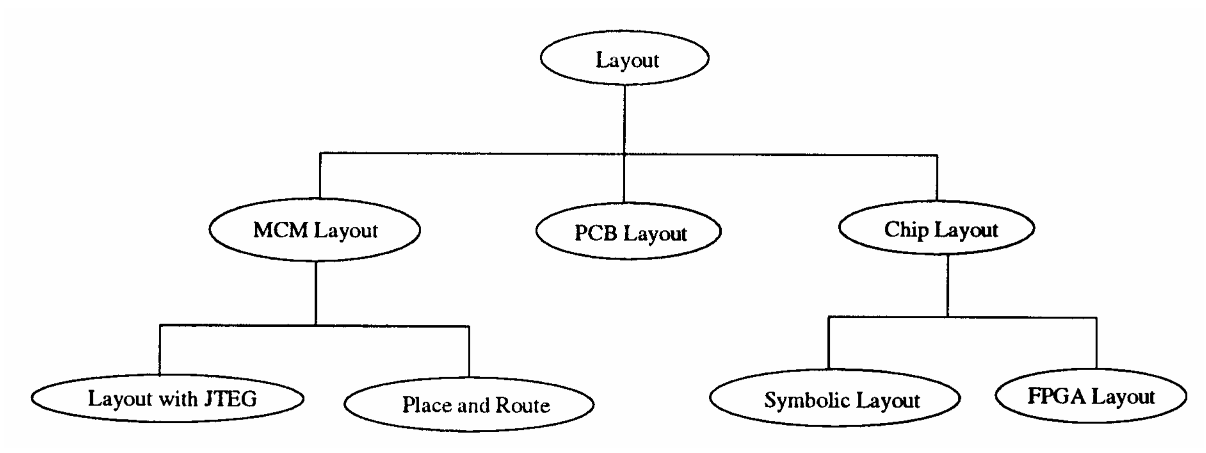


Figure 3. Layout Synthesis Task Hierarchy

3. The CAD Framework: Execution Environment

The CAD framework execution environment is a software environment which helps designers in selecting and executing design methodologies by allowing the systematic exploration of the design planning space. The execution model within the execution environment is modeled based on Petri Nets. When inputs are available, tasks are executed and outputs are created.

The execution environment allows backtracking, which occurs when the task cannot be accomplished. If the execution environment detects unsatisfactory results, the system is allowed to go back and try different alternatives. Another advantage of the execution environment is that it allows parallel exploration of the design space. Details of the execution environment are covered in Section 3.

In this section, the execution environment is discussed. The execution environment of the proposed CAD framework is modeled based on Petri Nest. A new approach to the execution environment, which dynamically constructs a process graph, automatically selects design alternatives, and automatically backtracks if the result is not satisfactory, is presented in this section. This section is organized as follows: First, an overview of the proposed CAD framework architecture is given in Section 3.1. The formal model of the execution model is explained in Section 3.2. In Section 3.6, the handling of multiple alternatives is explained. Other issues, such as constraints, process simulation, and version control, are covered in subsequent sections.

3.1 Proposed CAD Framework Overview

The proposed system is composed of several main components: Design Process Representation, Constraints, A Design Library, an Execution Environment or Cocpit, and Graphical User Interface (GUI). An overview of the proposed architecture of the CAD framework is graphically depicted in Figure 4. Design process representation represents design methodologies using the productions of a process grammar. Productions codify the possible hierarchical decomposition of tasks, which designers use to build a process flow graph. The process grammar naturally captures the hierarchical character of the design process and allows systematic exploration of the design space.

Design constraints are provided by the user. The system performs a preevaluation in order to select the best production or tool, and also performs a postevaluation after a task is finished. When preevaluation and postrevaluation processes are carried out, the system uses the constraints as input parameters. Constraints are items such as area, critical delay, die size, pin number, power consumption, etc. The Design Library contains various design data.

The execution environment program, Cockpit, keeps track of the design status and communicates with the designer via the GUI. The GUI helps users in several ways. Users can browse the available productions via the GUI and assign one or more input design data files together with their control information. The design progress can be displayed either in the form of a production or a Petri Net structure. The design path is displayed if the user chooses the history menu. Design data examination and a display of scoring results are additional features of the GUI.

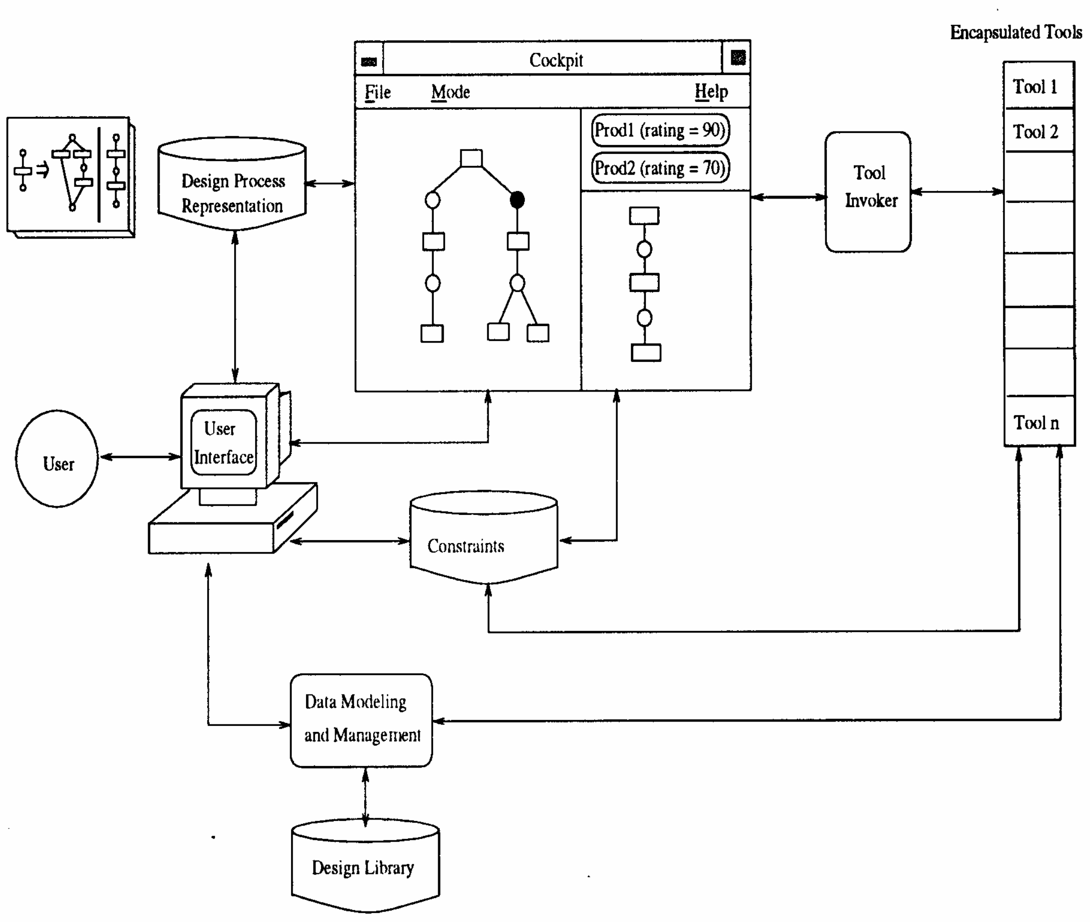


Figure 4. Block Diagram of System

3.2 Execution Model

The execution model provides for dynamic execution of tasks and the representation of state information. At a minimum~ the execution model allows the designer to access tasks and designs by tracking the information required to invoke tasks. The execution model constructs a process graph by selecting the proper production for each logical task. This selection is guided by invoking a preevaluation of the alternatives of the logical task. When an appropriate task is selected, the execution model either expands the graph or invokes a tool for the execution of a terminal task. After executing the tool, the execution model post-evaluates the result based on the criteria (constraints). If the result is not satisfactory, it backtracks to try another alternative.

The execution model is based on a Colored Petry Net and performs the following functions:

- Dynamic construction of a process graph.
- Preevaluation of the alternatives
- Selection of a production for each logical task.
- Execution of a tool.
- Postevaluation of the results.
- Backtracking if needed.

The execution model creates design flows by reading the production graph, determines the possible design alternative processes, and invokes the right tool for execution or expands the logical task. In order to choose the right alternative. Cockpit performs a preevaluation of all available logical tasks. An evaluation function is associated with each logical task. When the preevaluation and postevaluation processes are carried out, Cockpit uses the constraints as one of the input parameters.

Expansion is dynamically performed as the design process progresses. When the production graph is read by Cockpit and is converted into a corresponding Petri Net internal structure, a preevaluation function is called for each alternative and the results are posted, such as the score of each alternative, in the net. The highest score enables a corresponding transition. The scheduler in Cockpit now schedules or chooses which transition is to be fired based on resource availability. After finishing one path, the result is checked; this is called post-evaluation. If this does not agree with the anticipated result, the system backtracks to the selection point. and tries another alternative,

3.3 Cockpit

Cockpit is a routine of the execution environment which performs the following functions:
Creating daemon processes (initial graph).

- Keeping track of the design process.
- Dynamically construct a process graph.
- Scheduling task(s) by preevaluation.
- Performing the postevaluation.
- Interacting with the user.
- Controlling the GUI.

Cockpit is implemented using the algorithm described in Figure 5. Cockpit initially creates several daemon processes which maintain task specific knowledge. Cockpit's information about the design process comes entirely from an input file indicating a set of possible tasks and those decompositions that should be considered for each logical task.

The user interacts with Cockpit, which keeps track of the current status of the design process and informs the user of possible actions. Cockpit's display indicates to the user what design tasks have been completed so far and what tasks remain.

To assist the user in choosing an appropriate action, Cockpit invokes several evaluation functions. The evaluation functions provide ratings for the possible task decompositions and check the results. The ratings help the system to select tools. Cockpit determines what decompositions are available for the remaining logical tasks. This information is then displayed to the user.

Cockpit supports two modes of operation: manual and automatic. The manual mode is normally used for high-level decisions and stepping through the design process. The designer may wish to use the automatic mode for lower level decisions. In the manual mode, Cockpit waits for the user to select decomposition or execute a task. In this mode, the system performs preevaluation or a postevaluation and the system guides the user by showing the results. The user makes the final decision for selection of a production or backtracking based on the suggestions made by the system. When the user selects decomposition, Cockpit displays the new subtasks in place of the original task. When the user requests that a task be executed, Cockpit sends a message to the corresponding daemon process for execution. For terminal tasks, the tool invoker responds by invoking a tool. The user invokes the automatic mode by executing a logical task instead Qf selecting decomposition. In response to an execution message for a logical task, the daemon process uses encoded knowledge from a process graph to select a decomposition and then executes the sub tasks (also in automatic mode). If necessary, the designer may reverse any decision made by the daemon process in the manual mode.

In the automatic mode, the execution m0del utilizes the Petri Net structure more naturally since there is no human interaction. Cockpit dynamically creates design flows by reading the production graph, determining possible design alternative processes and invoking the correct tool for execution or expanding the logical task. In the execution environment, Cockpit uses the evaluation function to determine the alternative. After finishing the task execution, Cockpit postevaluates the result.

When the output of a task is not satisfactory, it is necessary to backtrack. Either different parameters must be supplied to some of the tools or different tools must be chosen; or alternatively, the task must be decomposed in an entirely different way. The designer may request that certain task decompositions be reversed. Additionally, if a decomposition was requested by a daemon process, that process can direct Cockpit to reverse it. Cockpit saves the state of the session before backtracking in case the designer later decides to cancel the reversal.

```
Initialization()
  {
    Start graph is selected;
    Create initial Daemon process and place tokens (send message);
  }
Wait for message;
    IF the message is from Execution process THEN
       {
          IF the message is WANT_EXPAND THEN
            {
               Invoke Pre-Evaluation function;
               Select Production based on Pre-Evaluation;
               Display expanded process flow;
               Send EXPAND_THIS or FAILED message to Execution process;
            }
          IF the message is POST_EVAL THEN
            {
               Post-evaluation;
               Send result POST_EVAL_OK or _FAIL to Execution process;
            }
          IF the message is FAIL_EXPANSION THEN
              Delete useless tokens;
          IF the message type is FAILED THEN
               Kill the child process;
       }
     ELSE /* Message from Daemon process */
       {
          IF the message is FAILED THEN
              Kill the child process;
          ELSE
            {
               Create a Daemon for the subsequent task;
               Put the output token in the newly created Daemon's input place;
            }
       }
     END IF;
```

Figure 5. Algorithm for Cockpit

3.4 Daemon Processes

Each daemon process is invoked (created) by Cockpit and execution process. Daemon processes are dynamic repositories of task-specific knowledge. Each message from the daemon process indicates the task being evaluated or executed and provides all the inputs and out- puts file names. The constraints may be included in one of the input files or may be passed to the daemon process directly.

Each daemon process is activated by an event signaling the arrival of a token in its input or output places. If an input event occurs, the daemon process creates an execution process by sending the task name to Cockpit to create the process. For an output event, the daemon process checks the output token numbers, which is assigned by the user. If the number of tokens does not reach the required number, the daemon process tries a yet untried alternative by changing the input token color. If more than enough tokens are generated, the daemon process selects the best tokens. Each daemon process retains the following information:

- Parent Task Name
- Name of input places
- Name of output places
- Child Task Names
- Information about the required output token numbers (counters)

The procedure for daemon process is shown in Figure 6.

```
Wait for messages;
IF the message is from its parent (Execution) process THEN
  {
    IF the message is TOKEN_N THEN
        Create a child Execution process;
        (Send the production name to the child process)
  }
IF the message is from its child (Execution) process THEN
  {
    IF the message is FAILED THEN
         Send the FAILED message to its parent (Execution) process;
    IF the message is TOKEN_IN THEN
         Send the TOKEN-IN message to parent (Execution) process;
  }
```

Figure 6. Algorithm for Daemon Process

3.5 Execution Process

Each execution process is created by a daemon and receives a production name. The execution process is responsible for invoking a tool, asking for expansion, and asking Cockpit to do a postevaluation after finishing its job. Each execution process handles only one token at a time. For multiple tokens, one execution process *is* created for each token. The execution process contains information about input places, output places, and its task name. The corresponding algorithm is shown in Figure 7.

3.6 Multiple Alternatives

Several alternatives may be simultaneously explored. This helps the user to obtain better results by selecting the best solution among several solutions. There are two forms of parallel exploration of alternatives in the design process: use of multiple parameter alternatives and multiple production alternatives. For a given production, there may be several parameter choices available. If a production does not produce an output which meets the design constraints, the same production should be tried with different parameter sets until all possible parameter sets have been exhausted. In addition, a given logical task may be accomplished in several ways. Each methodology alternative represents a separate production for the logical task.
Multiple alternatives can be expanded and executed concurrently if the user specifies multiple tokens in the logical production's input place and/ or output place via the QUI. The system assumes one token is in each place if the user does not place any token in the input or output places of a production. There are two cases which should be considered for multiple alternatives:

1. Multiple Tokens in the Input Place:

The number of tokens indicates the number of productions to be simultaneously executed. If there is a child production which also has multiple tokens, it is carried out simultaneously. The total number of productions active at any given time is controlled by a global control variable, *Total Production*, and by resource constraints.

2. Multiple Tokens in the Output Place:
The number of tokens indicates the number of desired acceptable outputs. If the number of acceptable outputs reaches the token number, the production is considered a success. If not, the system backtracks and tries other productions. If the desired number has not been reached even after all the productions have been tried, all acceptable outputs are used for the next step

Basically, the number of multiple token in the output place dictates the number of alternatives that must be tried unless the same input token is used as input to different transitions. These aspects are illustrated in Figure 8 and Figure 9.

. Then multiple alternatives are executed simultaneously and several compatible outputs are produced, the system must select from among the requested number of outputs. Selection is based on the chosen selection strategy, first-available (FA) or best-choice (BC).

```
Execution()
   {
     IF Terminal task THEN
         Execution of the terminal tool;
         (Send TOKEN_IN message to parent Daemon process)
     IF Logical task THEN
        {
           Send WANT-EXPAND message to Cockpit;
           Wait reply from Cockpit;
               IF the message is EXPAND-THIS THEN
                  {
                     Expand the graph by creating one Daemon process for
                        each task node in the production;
                     Send TOKEN-IN message to the child Daemon process;
                     Wait for messages from these children Daemon processes;
                     IF the message is from child Daemon process THEN
                         {
                            IF the message is TOKEN_IN THEN
                               {
                                  Send POST_EVAL to Cockpit;
                                  Wait for reply from Cockpit;
                                  IF POST_EVAL-OK THEN
                                     Send TOKEN_IN to parent Daemon process;
                                  IF POST_EVAL_FAIL THEN
                                     Send FAIL_EXPANSION, FAILED, WANT_EXPAND to
Cockpit;
                               }
                            IF the message is FAILED THEN
                              {
                                  Send message to parent Daemon process;
                                  Exit;
                               }
                           }
                    }
                IF message is FAILED THEN
                     Send FAILED to parent Damon process and Exit;
          }
    }
```

Figure 7. Algorithm for Execution Process

A problem can occur when multiple alternatives handle the output files. Since each alternative production creates an output and the file names are the same for all alternatives, overwriting to an existing output file should be prevented. To solve such a problem, different working directories are used for each token.

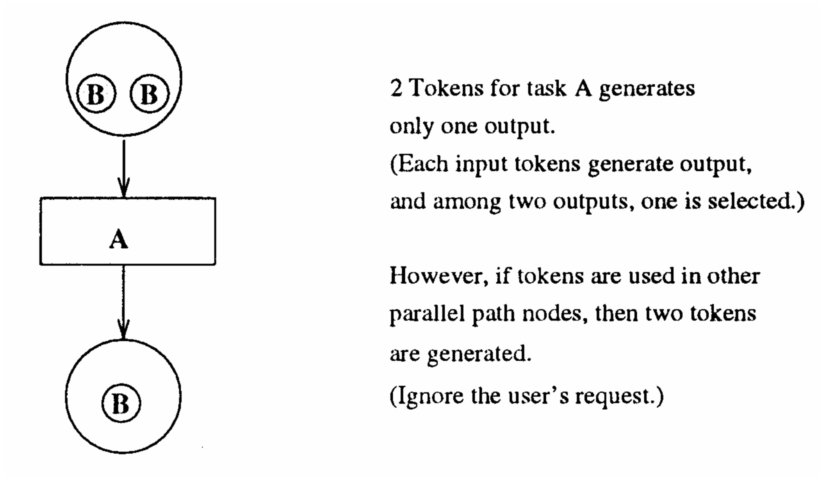

B
B
A
B
2 Tokens for task A generates
only one output.
(Each input tokens generate output,
and among two outputs, one is selected.)
However, if tokens are used in other
parallel path nodes, then two tokens
are generated.
(Ignore the user's request.)

Figure 8. Smaller Number of Tokens in the Output

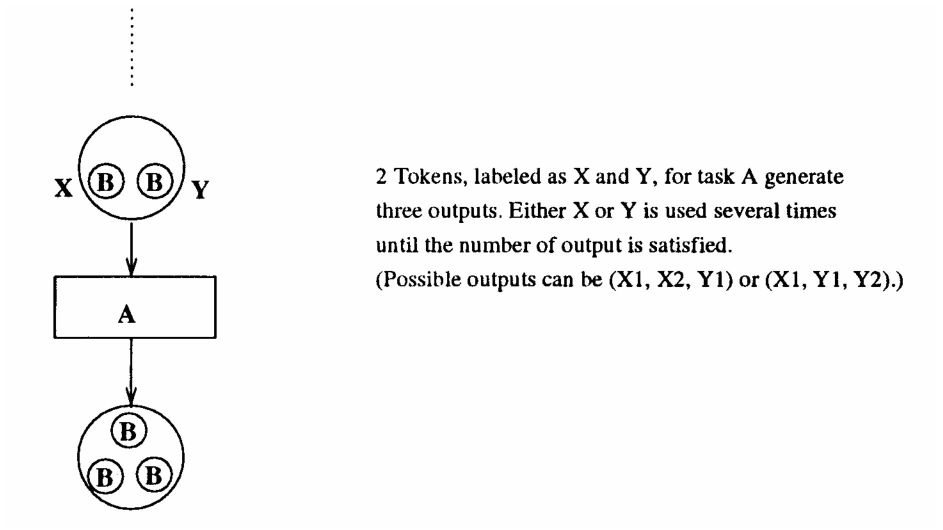

X
B
B
Y
A
B
B
B
2 Tokens, labeled as X and Y, for task A generate
three outputs. Either X or Y is used several times
until the number of output is satisfied.
(Possible outputs can be (X1, X2, Y1) or (X1, Y1, Y2).)

Figure 9. Larger Number of Tokens in the Output

3.7 Process Simulation

Based on the evaluation results, the execution environment makes suggestions as to which production and/or tool is best suited for the given input. Using these functions, together with all the values assigned to each production design process simulations are possible without actual design process invocation.

The input file type, the file size, the maturity of tools and productions in the CAD community, and estimated time to finish a task given by the input file comprise several examples of parameters the evaluation function can use to determine the suitability of the production. Design process simulation allows the user to predict or expect certain results.

3.8 Graphical User Interface

GUI is used to establish communication between the user and the execution environment. Through GUI the user can do several things, such as set the initial graph, partially expand the process graph, and browse through the alternatives.

The Production Editor Main Window is used to create, browse, and edit productions as well as to edit task node specifications and specify input/output node information. A top view of the editor window is shown in Figure 10.

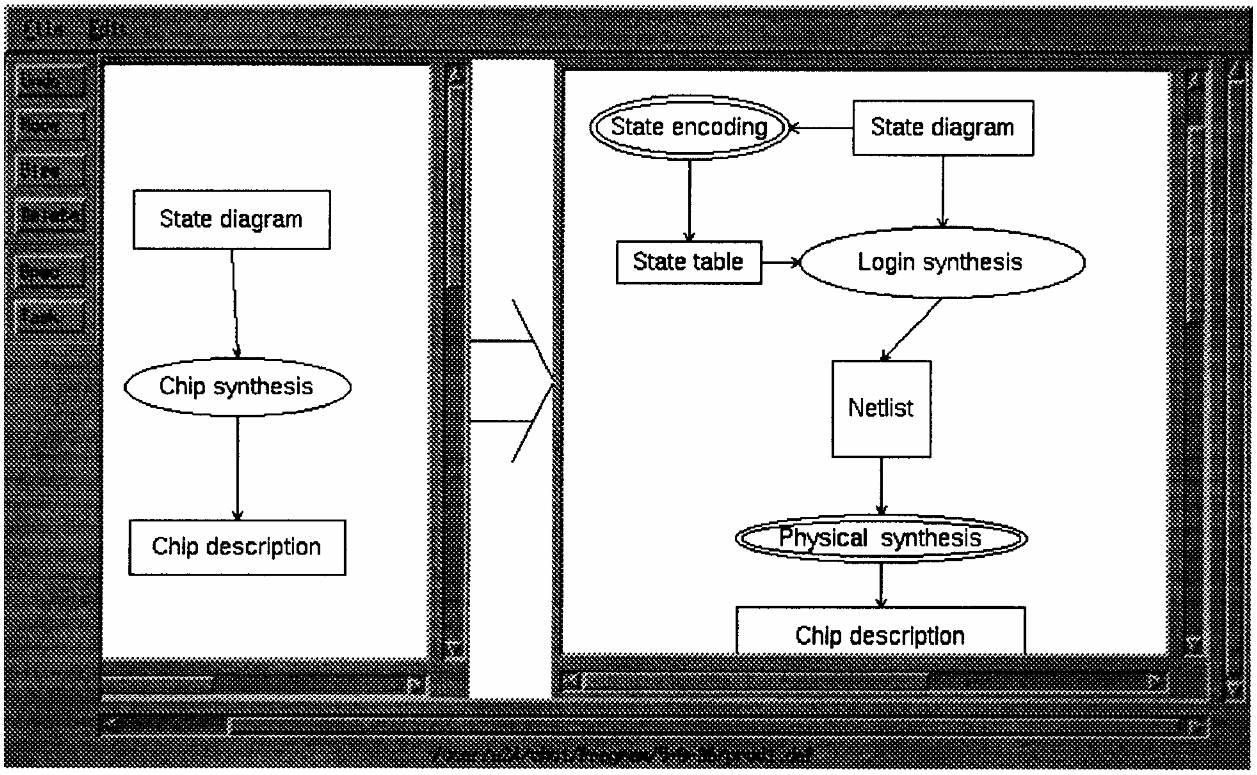


Figure 10. Production Editor Window

In the case of a rollback, the display of the situation is as follows: First, the rollback message is displayed at the bottom of the Message window, ensuring that the user can see what has happened in the system. Then, the parent production graph (the graph displayed just before the expansion leading to this task) is redisplayed and the same preeva1uation process is invoked.

Cockpit must record this history; that is, when the user requests the execution history, the QUI displays the overall history using different colors, e.g., a failed path is drawn with red lines and a current path/success path is drawn with blue lines.

3.9 Constraints and Checklist

Constraints are used to select a proper tool for a given task, to execute the tool, or to verify the correctness of a design. Constraints must be managed properly so that the CAD framework can function properly. Area, maximum/minimum delay, power consumption, pin number, operating condition, maximum fanout, wire load, clock period, technology library, and testability requirement are several examples of such constraints in the computer hardware design.

Kim [2] categorized constraints into four different categories: performance constraints, environment constraints, relativity constraints, and selection constraints. Some examples of performance constraints are area and delay; operating conditions are environment constraints. Relativity constraints restrict what other designs can be used in conjunction with a design when it is instantiated as a component, while selection constraints restrict what designs can be instantiated for a particular component of a design. This classification of constraints is helpful for analyzing characteristics of the constraints themselves,
Baldwin introduced a new language to express constraints [6]. Although this language has been claimed to be powerful enough to express any kind of constraint, it has its drawbacks, Designers must learn the language syntax to express constraints, not a simple task for hardware designers.

Kim [2] and Baldwin [6] considered constraints associated with design data only. However, in order to form a good CAD framework, there should be some way of answering a question like "Which tool (or program) produces a better result for a given input?" These kinds of constraints, tool selection or production selection constraints, should also be handled. Several examples of such constraints aye tool release history, size of the tool, average execution time, and user's preference. These constraints are used by preevaluation functions.

The quality of a design result depends on the selection of tools, design methodology, and design data from certain design libraries. Each tool has different qualities or capabilities, such as maturity of tools, the speed needed to produce output from given input, and the output quality produced using the given input data. Each designer can define any variable for a production and assign/modify a value in an ASCII format. An example of such definition is shown in Figure 11. When a production Arch_Syn is applied, the user uses a preevaluation function, named "preeval1", in the current working directory. This routine is written by the user and precompiled. A postevaluation routine can also be defined as well. The next three lines consist of actual variables and values assigned by the designer .

```
Arch_Syn.PRE ./preeval1
Arch_Syn.POST ./hello
Arch_Syn.0 time 9 pref 3 history 4
Arch_Syn.1 time 11 pref 2 history 5
Arch_Syn.2 time 4 pref 6 history 2
```

Figure 11. Production Scoring Example

Designers write preevaluation functions using these values. For example, a very simple but complete preevaluation function is shown in Figure 12. Here, if the designer assigns different weights to the variables *t*, p, or *h,* a different evaluation result is produced, where *t* represents the time to finish this production, *p* represents the variable which holds the penalty value for converting the input file type, and *h* represents the time the production has been available. The weights are assigned by the designer based on experience or preference. In this example, the designer prefers a long history of the production and shows very little concern about the translating file type.

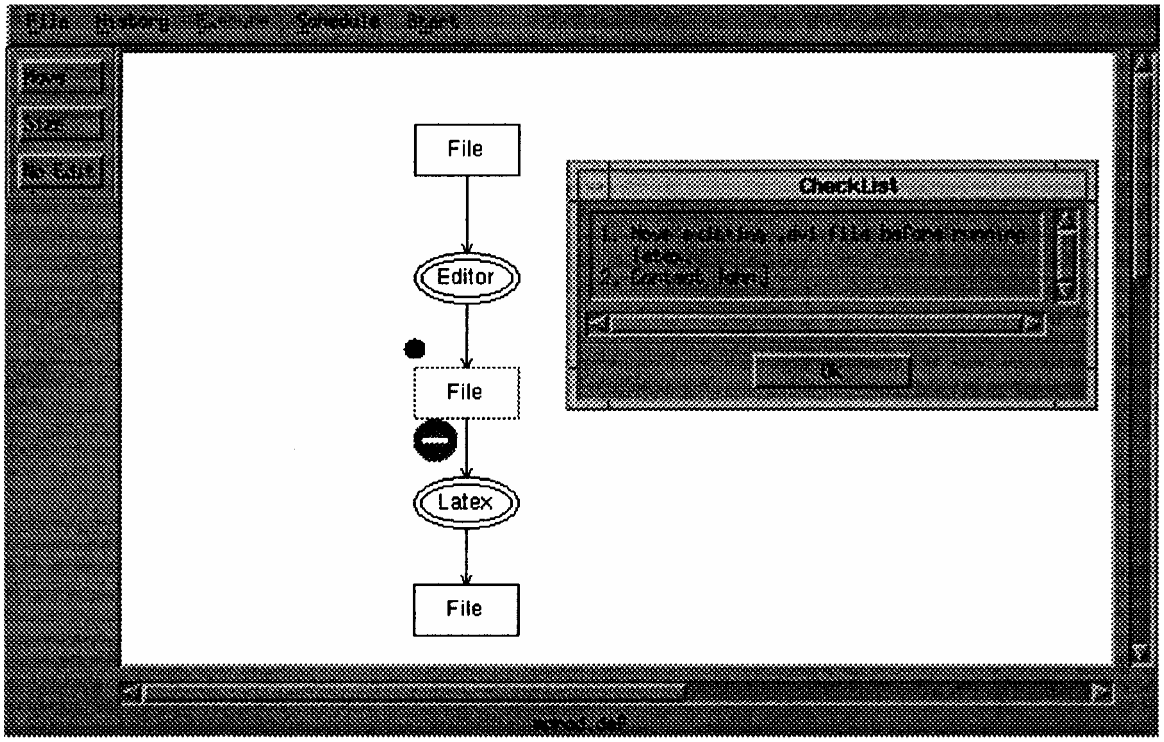
Similar functions can be written for postevaluation functions. After each production is completed, the postevaluation function is invoked and determines whether to accept the result or not.

```
#include <stdlib.h>
#include <stdio.h>
main(int argc,char **argv)
{
int t,p,h;
int score;

if (argc < 7 )
  exit(-l);
t=atoi(argv[2]);
p=atoi(argv[4]);
h=atoi(argv[6]);

score = t*0.2 + p*0.l + h*0.7;

exit(score);
}
```

Figure 12. Preevaluation Function Example

In this way, the specification and the execution environment can actually be separated. Different designers can also use different ratings without modifying the productions.

The checklist is a utility similar to “reminder”, in which a checklist can be created by the designer. When the design process reaches a predefined point, the designer can browse the contents of the checklist. This feature is not directly related to the actual execution environment. The checklist helps the designer remember things that must be done. The breakpoint feature can help the system stop at a certain design point where the checklist can be examined. An example is shown in Figure 13.



1. Move existing .dvi file before running latex.
2. Correct latex]

Figure 13. Break Point and Checklist

3.10 Resource File

Resource files are used to describe the details of each tool. These files contain information such as runtime parameters, environment variables, preconditions, postconditions, the full path name where each tool is located, input/output requirements, and machines where the tool can be executed. Here, a *makefile* format is adopted. This resource file consists of two parts: The first part is a macro definition section and the grammar is:

var = definition.

The definition must be written in one line. For example, the VHDL simulator, *vhdlan,* can be defined together with its path name as

VHDLSIM = /home/pixel/146/synopsys/sparcOS5/sim/bin/vhdlan

The second part is an execution file definition section and its grammar is

exe_file : definition.

The definition can be written in multiple lines, where each line corresponds to a command. For example: a macro definition of a process which consists of *emacs* editor execution and then *cc* is

```
emacs-then-cc : emacs $(I0)
           $(CC) -c $(I0)
          O0 $*(I0).o
```

At any time, $(var) can be used to fetch the value of a macro previously defined; the predefined macros I0 and I1 are the inputs to the execution file No.0 and No.1 (*emacs* and *cc*), respectively. O0 filename means "filename" is used as the first (No.0) output file with the new extension .o in this example.

For distributed environment, a line is added before any regular command and O* command.

```
VHDLcomp:   Use localhost samisen calliope musette
            vhdl2xnf $(I0)
            O0 $*(I0) .xnf
```

where Use <machine> line tells the program to execute the commands following the line on the given machine. There is no default machine, so even if the user wants the program simply to be run in a local machine, the user must still explicitly write the line Use localhost.
The main advantage of using this file is that even if the system environment is changed, it is not necessary to recompile the system software since each tool description is not hardcoded' in any of the software. Only the resource file should be changed.

3.11 Load Balancing

In a distributed environment, load balancing is one of the most important issues in system performance. All system performance depends on resource contention. In any computer system there are three basic resources: CPU, memory, and the input/output (IO) subsystem. Among these three types of resource usages, CPU usage is the main concern because most CAD tools are CPU intensive.

Each process (or program) requires a certain number of CPU cycles to execute, and it is not possible for a single process to use the CPU alone until execution is finished. Usually, several processes share the CPU at one time. If loads are assigned to machines which are already heavily loaded, then the overall system performance is degraded: processes in a heavily loaded machine take a long time to finish) and the remaining tasks may depend on the results of the previous processes.

There are several ways to measure CPU contention. The simplest one is the UNIX load average, reported by the *rup* command, which shows the host status of remote machines. The load average tries to measure the number of active processes at any given time. A typical result of this command is

```
pixel     up 12 days,     6:38,     load average: 0.23, 0.19, 0.01
```

The first load average (0.23) is measured over the last minute. The second and the third load average are measured over the last 5 and 15 minutes, respectively.

In the proposed systemy the machine with the smallest load average at the time of task execution is used. Available machines are listed in the resource file. This at least ensures that a particular machine is not overloaded before assigning it a task. The selection criteria can be extended by examining the second and third load averages, from which the load trend can be inferred.

Several problems remain associated with the method described above. First, the command *rup* does not guarantee the correct result. For example, if the Network File System (NFS) server crashes while a process is waiting for the disk IO to complete across NFS, the process is considered to have been running the entire time although nothing was actually happening. Another problem is that the load average does not account for priority. Finally, the load average cannot predict future events.

4. Routines

4.1 Cockpit

Global variables

DLList<MProduction *> prod_db; / production database */

DLList<Token *> tokenList; /* token list */

DLList<BreakPoint > bpList; /* breakpoints list */

DLList<CheckList *> ckList; /* list of checklist */

MExtGraph *cpGraph; /* the graph and expansions in the main window */

MGraph goGraph; /* the final graph in the main window */

int ExecutionMode; /* the execution mode, manual or auto */

MSpecNode *work_sn; / the specnode the pointer is pointing at */

MTaskNode work_tn; /* the tasknode the pointer is pointing at */

User interface part functions (call backs)

Cockpit-FileCB:
  callback function for file menu:
    open:
   call yyparse() to read in production file.

Cockpit-EditCB:
   callback function for edit button on left panel:

Cockpit-TokensCB:
   callback function for token menu:

Relationship between functions in Cockpit can be shown as in Figure 14.

4.2 Daemon

daemon.c

For each group of tokens, the daemon will spawn an execution process to do the calculation.

The group names are stored in exec_name[][], input tokens are stored in exec-in-tokens[] and output tokens are stored in exec_out_tokens[]. *TCP* socket that *is* used to communicate with child execution process is stored in ch_soc[], socket is set to MWAITING~TOKEN if the input tokens are still not complete, set to MNON-EXIST if the child execution process is dying.

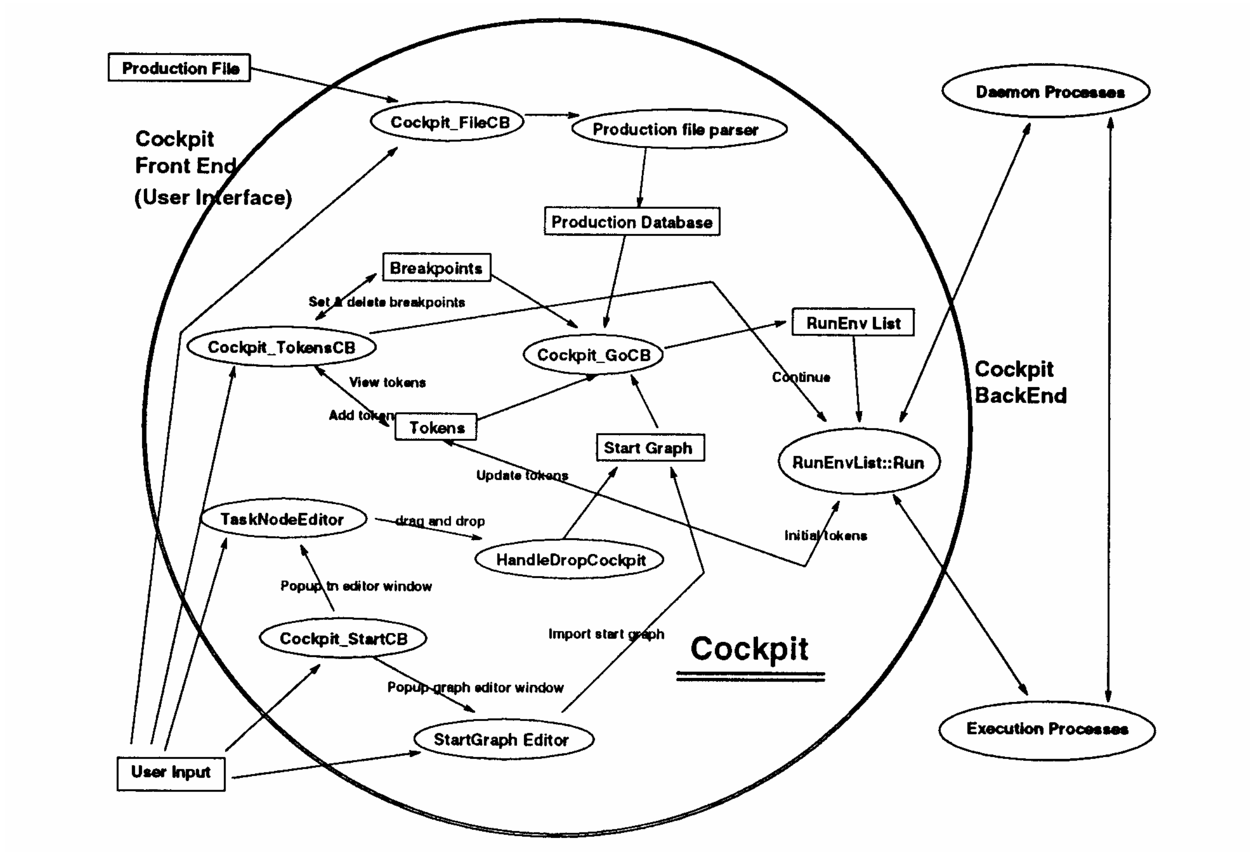


Figure 14. Relationships between Functions

l. Daemon initializes.

2. Daemon waits messages from its parent execproc:

- TERMINATE: daemon will exit after this message.

- .TOKEN-IN: receive token from parent, the token group name is checked to see if there is already a record of that name in exec_name[], if so, the token is added, if there is enough tokens in this group to fire the transition, a child execproc will be generated to do the calculation. If there is no such group name record in exec_name[] , a new record is made and the corresponding Ch-soc[] entry will be set to MWAITING_TOKEN.

- TOKEN-OUT: we actually do nothing here.

3. Daemon waits messages from its child execproc:

- FAILED: Child execproc has failed, in this case daemon will report this message to its parent execproc and then mark corresponding ch_soc[] to MNON-EXIST. The connection to this failed child is no longer used.

- TOKEN~OUT: Child execproc is using the input t6ken. We send a TOKEN_OUT message to parent execproc (if there is any).

- TOKEN_IN: Read in this token, store it, and wait until there are enough tokens for this group, then report this group of output tokens to the parent execproc. Close the connection (set ch_soc to MNON-EXIST).

4.3 Execution Process

execproc.c

1. execproc initialize: Get node info and all the input tokens. If the tasknode is a terminal node, then take away the input tokens, (send TOKEN_OUT to parent daemon, GUI_TOKEN_OUT to GUI). Find out the commands we are to execute from the resource file, execute them, then report the output tokens
(send TOKEN_IN to parent daemon, GUI-TOKEN_IN to GUI). If the tasknode is a nonterminal node, then send WANT_-EXPAND to GUI, asking for an expansion of this nontermina1 node. Then wait for reply from GUI. The reply can be be either EXPA.ND_THIS, or FAILED. If it received EXPAND_THIS, then read in the necessary information. , Execproc does the expansion by creating one daemon process for each tasknode in the expansion graph and then waiting for messages from these child daemon processes:

- FAILED: There is nothing to expand, send FAILED to parent daemon process and then exit.

- TOKEN_IN: This message means some child daemon has generated outputs, so the execproc reads inn this token, then sees if this token should be sent somewhere. This is done by looking at the expansion graph. Send this token if necessary, and then look if this token is put into a specnode that is :mapped to a output specnode of the original tasknode (the tasknode before the expansion); if it is, then then store this token in out_tokens. If there are already enough tokens, then send message POST_VAL and those output tokens to GUI for postevaluation. Jhe reply can be POSTEVAL_OK or POST-EVAL_FAIL. If it is POSTEVAL_OK, we report these output tokens to parent daemon process (send TOKEN_IN). If it is POSTEVAL_FAIL, this means the expansion we have chosen didn’t give us the correct result, so we have to choose another expansion (if possible). We do this by sending FAIL_EXPANSION to GUI. The GUI will fail the current expansion and delete the useless tokens. The execproc will then send WANT_EXPAND again to GUI to see if there is another possible expansion.

- TOKEN_OUT: This message means some child daemon is using its input token. If this input token is mapped to an input token of the original tasknode, then we send TOKEN_OUT to the parent daemon. Otherwise we do nothing.

5. Data Structures

```
class MgraphObject    /* base class of MSpecNode and MTaskNode */
  Data:
      Widget widget;
       /* the widget the object is attached to, should be NULL is no widget,
          otherwise UnsetWidget() will fail */

       String name;  /* the name of the object */

       int x,y,width,height;  /* the position and size of object */

       MGraph *graph;   /* the graph who own this object */

       char pathName[];   /* the pathName is the MExtGraph */
        int hasToken;    /* set to True if there is token is this object */

Method:
   Constructor:
   MGraphObject();
   /*
        this should never be called directly, it's supposed to called by
        the constructor of MSpecNode or MTaskNode.

        widget = NULL, graph = NULL, pathName[0}=0; hasToken = 0;
   */

   int IsA(); /* return the type of this object, can be GENERAL_OBJECT,
                         SPEC_NODE, or TASKNODE */

   GetName(), SetName(), GetX(), GetY(), SetX(), SetY(),
   GetWidth(), GetHeight(), SetWidth(), SetHeight();

   SetWidget(Widget w);
   /*
        this will attach the object to the given object w, if it is already
        attached to one, the old one is detached (UnsetWidget), the widget's
        XmNuserData is used to store a UserData structure, which is used to
        by many functions as ObjevtEventHandler.
        ObjectRedraw will be added as XmNexposeCallback callback.
        ObjectEventHandler will be
        installed as an event handler of the widget, the event handler will
        handle the move and resize of the object.
  */

  UnSetWidget();
  /*
       detach the object to widget, remove event handler and callback.
  */

  virtual functions:

  Draw();
  Undraw(); /* ??? */
  int isInside(int mx,int my);
  int FindIlntersect(int mx,int my,int *rx, int *ry);

       /*
          suppose we want to draw a line from the object to a point (mx, my),
          this function will figure out from which point ( returned by rx, ry )
          should we start to draw this line.
          return value is True if mx, my is outside of object,  False if
          mx, my is inside of the object.
       */

       AddEdge(int type,MEdge *); /* attach an edge */
       DelEdge(int type)MEdge *); /* detach an edge */

/*
* about edges:
* edges always goes from a specnode to a tasknode or from a tasknode
* to a specnode, when an edge connect a specnode and a tasknode together
* they must match, i.e. the tasknode must have corresponding named
* in(out)put.
* function IsObjectsConnectable can exam if two objects can be connected
* to each other.
* a specnode can have more than one fan-out, but it can only have
* fan-in 1. a tasknode has fan-in equal to its GetNumInput(), fan-out
*' equal to its GetNumOutput () .
*/


class MSpecNode ( inherit from MGraphObject )
    Data:
         DLList<Edge *> inEdgeList; /* there is only 1 edge is this list */
         DLList<Edge *> outEdgeList;
    Method:
      Constructor:
          MSpecNode();
          MSpecNode(SpecNode *);
          MSpecNode(const MSpecNode &);
          MSpecNode(char *name,int width,int height);
          MSpecNode(char *name);
    Destructor:
        ~MSpecNode();
            /*
                all edges in inEdgeList and outEdgeList are DESTROYED)
                UnsetWidget,
                if it's owned by a graph, call graph->DelObj.
            */

            MTaskNode * find_input_node();
            MTaskNode * find_output_node(MTaskNode *after);

        /* this function is to be used to traverse the outEdgeList

              this is needed because in daemon and execproc, the token's location
              is given by (tasknode, place_i, place-type), in gui, we use specnode's
              path name to represent token's location. we need to do the type
              conversion.
              path name -> (tasknode, place-i, place_type)
                    find the MSpecNode by find_pathname-nth, then find_nth_obj.
                    call tn = find_output_node to find the connected tasknode.
                    call tn -> find_node_nth to find place_i.
        */


class MTaskNode ( inherit from MGraphObject }
  Data:
     TaskNodeDef def;
     /* the definition of this tasknode, include information about inputs
          and outputs */


     MEdge * inEdgeList[] ;
     MEdge * outEdgeList [] ;
     /* the reason to use array to implement in(out)EdgeList in MTaskNode
          is it's frequently used to use in(out)EdgeList[i] to fetch the
          ith in{out)put edge. */
    Method:
     MTaskNode();
     MTaskNode(char *);
     MTaskNode{char *,int width,int height);
     MTaskNode(TaskNodeDef *);
     MTaskNode(const MTaskNode &);
     /* copy constructor, the edge list is not copied, because it
          doesn't make any sense to do so */

     int GetNumlnput();
     int GetNumOutput();
     int find_edge_nth(MEdge *, int type);
```

```
        MEdge *find_nth_edge(int n, int type);
         /* return ( type == M_IN_EDGE ? inEdgeList[n] : outEdgeList[n] ); */
        MSpecNode *find-nth-obj(int n,int type);
         /* return (MSpecNode *)( type == M_IN_EDGE ? inEdgeList[n]->startObj :
                                    outEdgeList[n]->endObj ); */


class MGraph
Data:

Widget widget; /* widget the graph is on */

DLList<MGraphObject *>objList;
/* list that contain all the objects */
DLList<MEdge *> edgeList;
/* list that contain all the edges*/

Method:
/* Create, Copy, Delete */
MGraph(); /* create an empty graph */
MGraph(const MGraph &g); /* create a copy of graph g */

/* find index of object, and */
int find-obj-nth(MGraphObject *); /* return -1 if unsuccess */
/* find index of object given path name */
int find-pathname-nth(char *pathname); /* return -1 if unsuccess

void Shift(int sx,int sy);
/* shift the graph by sx,sy, when we expand the graph, sometime the graph
grow out of the boundary, we might need to shift the graph */
void FindBoundBox(MBoundBox *bbox,MMap *map = NULL);
/* calculate the bounding box of the graph, the mapped specnodes are not
counted, result is stored in bbox */

void Expand(int i, MGraph *g, MMap *map);
/* expand the ith node with graph g, with map */
/* this function is called by MExtGraph::Expand() */

}


class MProduction {
Data:
MGraph leftGraph;
Widget leftWidget;
Widget rightWidgets; /* manager widget that contains all right side widget */
DLList<GraphWinData> rightSide;
Method:
MTaskNode *GetLeftSideTaskNode(); 1* get the tasknode of the left side */
```

```
void AddAlt(MGraph *,const ProdInfo &)
/* add this graph and produc;tion info as one of the right side of production */
void DelAlt(MGraph *)
/* delete one right side */

void SetWidget(Widget lw,Widget rw)~
/* left side widget draw in lw, right side widgets are created as children
    of rw */
void UnsetWidget();
}

/*
purpose:

start from a initial graph, given several expansions, we are to determine
the expanded graph, and we are able add a new
expansion, the new expansion can be based on the old one, e.g. we first
expand node "1" to "1.1",  "1.1", "1.2". then after that we can add a
new expansion for node "1.1". We can also delete an expansion, node
deleting expansion for node "1" will automatically delete expansion f or
node "1.*".

To distinguish a node from others, we use "pathname".
in each graph, there is a index number for every tasknode and specnode.
we concatenate these index number together with '.' in between.
*/

class MExtGraph {
Data:
Widget widget;
MGraph start_g; /* the graph we start from */
DLList<MExpansion> expansionList;
MGraph end_g; /* the result of expansion */
Method:
}

/*
purpose:
a data structure to hold information for a group of token to run and display.
to construct a RunEnv, we need the name of token, a parent Widget,
an initial graph, a production list, a breakpoint list, a token list.
*/

class RunEnv {
Method:
constructor, destructor,
AddToken(Token *);  /* add a token to the token list */
HandleInput(int soc);
/* handle information from a communication socket, this function is always
                                        called by a RunEnvList's Run() function */
```

```
}

/*

purpose:
data structure for SEVERAL group of token to run and display.
RunEnvList will create several RunEnv for each group of tokens it has.
*/

class RunEnvList {
Data:
Widget pShell;
/* this widget should be the cockpit’s drawarea */

RunEnv *envList[]; /* array of RunEnv, one for every group of tokens */
int n_group; /* how many groups */
char idList [] [];  /* the name of each group */
int dpy_mask[];  /* if want to show on screen, True, otherwise, False */
int n_daemons; /* how many daemons */
int daemon_soc[] ; /* each daemon's socket */
int daemon_node[] ; /* each daemon's node */
int node_daemon[] ; /* the daemon number for node */
XtInputId daemon_inputId[];

/*
* after we create a socket for a daemon,
* we need to call XtAddInput to add a callback to monitor the
* socket, so we record the return value of XtAddlnput to this
* array so that we can remove this callback later.
*/

/*
Method:
to construct a RunEnvList, we need a parent widget, a initial graph,
a production list, a breakpoint list, a token list.
*/

Run(); /* run this */
Cont (BreakPoint bp);
SelectShow(); /* set dpy_mask */
}
```

6. Production File Grammar Specification

```
file:      SPECDEFFILE specnode_defs |
          TASKDEFFILE tasknode-defs |
          PRODUCTIONFILE tasknode-defs productions |
          GRAPHFILE tasknode-defs graph;

     specnode_defs: specnode_def | specnode_defs specnode_def;

     tasknode_defs: tasknode_def | tasknode_defs tasknode_def;

     productions: production | productions production;


parent: /* empty */ I PARENT QSTRING;
specnode_def: SPECNODE QSTRING /* node name */ parent;
tasknode_def : TASKNODE QSTRING parent inputs outputs
terminal exefile preevalfile preevalref postevalfile postevalref
terminal: TERMINAL INT; /* terminal or non-terminal node */
exefile: /* empty for non-terminal node */ I
              EXEFILE QSTRING; /* execution file for terminal node */
preevalfile: /* empty for terminal node */ I
              PREEVALFILE QSTRING /* pre-eval file for non-terminal node */
preevalref: /* empty for terminal node */
               PREEVALREF QSTRING /* pre-eval reference file for non-terminal*/
postevalfile: /* empty for terminal node */
               POSTEVALFILE QSTRING /* post-eval file for non-terminal */
postevalref: /* empty for terminal node */ J
                POSTEVALREF QSTRING /* post-eval reference file for non-terminal*/
inputs: input | inputs input;
input: INPUT QSTRING; /* input specnode name */
outputs: output | outputs output;
output: OUTPUT QSTRING; /* output specnode name */

productions: production | productions production;
production: PRODUCTION LEFT graph rightgraphs;

rightgraphs: rightgraph | rightgraphs rightgraph;
rightgraph: RIGHT graphinfo graph;
graphinfo: /* empty */  | HISTORY INT SIZE INT PREF INT;

size: width height;
width: WIDTH INT;
height: HEIGHT INT;

position: POSITION point;
point: '(' INT  ','  INT ')';
graph: GRAPH index_nodes edges;
```

```
index-nodes: index-node | index_nodes index_node;
index_node: INT node;
node: specnode | tasknode;

specnode: SPECNODE QSTRING position size;
tasknode: TASKNODE QSTRING position size;

edges: edge | edges edge;
edge: EDGE FROM INT TO INT intpts;
intpts: /* empty */  | intpt | intpts intpt;
intpt: INTPOINT point;
```

7. How to Use

7.1 Cockpit

Prepare to start:
1. Read in a production file. The production file can be read in by File...Open menu, or specified in the command line "-p" argument.

2. Move something to the main window as the start graph. Start graph can be either a single tasknode, or a graph consisted of several tasknodes and specnodes. If you want to start with a tasknode, choose Start...Tasknode, a tasknode editor window will appear, it's the same window as the one in production editor, so you can use drag and drop to move a tasknode into the main window. The other way is to start with a preedited start graph; by using the Start...StartGraph menu, a graph editor window will popup and let you edit start graph as you wish. The start graph editor's usage is very similar to how you edit graph in the production editor window. Please see production editor section for reference. After you've made a sta~t graph, use File..Start menu in graph editor window to put the start graph into main window, then you are ready for the next step.

3. Add some tokens to some input places. To fire a network, we need some input tokens. Before starting to execute the model, put some input tokens into some input places. As for what token should be put and where these tokens should be assigned, please refer to the concept section. There is a popup menu in the main window. Press the left mouse button when the pointer is pointing at the specnode where the token should go. From the popup menu, select Tokens...Add, and then answer the question dialog about the information of the token. The token will be put in that specnode. The specnode is green when there are any tokens in it and black when empty. The tokens are shown above the left top corner of the specnode as a black dot. Find out how many token are in each specnode by just looking at the graph. To examine the content of a token, use Tokens...View from the popup menu.

4. Set the in and out value for each tasknode. If adding more than one group of tokens to the start graph, then it's necessary to change the in and out value for some tasknode. The in and out values te11 the computer how many input token groups the tasknode will be expecting, and how many output token groups it will generate. For example, if a tasknode has in=2, out=l, then it will wait until it has 2 groups of tokens in its input place. Then it will invoke the execution process to generate 2 output tokens. Finally it will select one of them. You can't see the out value since it's undefinable. To change the in and out value for a tasknode, click the left mouse button on the tasknode to access the popup menu. Select " In & Out" and then fill in the blanks.

5. Set breakpoint if necessary. Breakpoints can be set if it's necessary to examine the output tokens generated in the intermediate step. Each breakpoint is associated with a specnode. Click the left button on the specnode you want to set breakpoint and choose Tokens...Break, Then a red stop sign will be underneathe reft corner of the ,specnode, showing that there is now a breakpoint. Execution will be stopped when new tokens arrive in this specnode. When the execution is stopped, continue by choosing the Token...Cont menu.

Start:

Click the menu “Execute-Manual" or “Execute-Auto".

The user will be asked to choose which group of tokens he wants to show on the screen. Each token group will use a separate window to perform execution and expansion.

The graph will be executed and expanded. When a terminal tasknode is fired, the corresponding execution file will be executed (according to a resource file, which will be explained later) when a non-terminal tasknode is fired. If manual mode is selected, the user will be prompted to select an expansion graph from several alternatives. A popup dialog will show all available options, the preevaluation scores of these options. If any option has is already tried and failed, it’s score is “FAILED". If there is no untried option, press “Cancel.” Otherwise select the production number, then press “OK”. If auto-execution mode is selected, the right side with the largest preevaluation score will be selected automatically. When the expanded non-terminal node gets tokens in its output place, it will perform a postevaluation.

It’s necessary to run a *rdaemon* in the given remote machine. This is IMPORTANT because the program has to be able to communicate to the remote daemon to send files and commands. "The sequence is as follows:

1. Start *rdaemon* on the remote machine first. If you want the remote machine to display x-windows on the local machine, enter “setenv DISPLAY" before starting *rdaemon.*

2. Then start *cockpit*.

7.2 Editor

Within the editor window, the user is allowed to create or modify productions and edit specifications. When the command editor is issued, an editor window is shown. There is one window for a left side of a production, and will be 0, 1, or more windows for the right side in the editor window in which the current production is displayed. Editor keeps a production database in memory, thus user can only edit one production (current production) at any time.

- Left Side Window: Only tasknodes can be added to window (using drag and drop which are explained later). After a task node is added, all its input and output specnodes will be added automatically since input and output should be matched. The user may move or resize them.

- Right Side Window: Tasknodes and specnodes can be added to a window (using drag and drop). Edges can be added use left-most mouse button: click the start object first, click any intermediate point if any, then click the end object. Move the mouse outside window when finish adding. If it is an invalid connection, (e.g., doesn't match the node definition) is tried, the system does not allow it and gives a beep sound. Tasknodes and specnodes can be moved, reslzed, and deleted. Edges can be deleted.

7.2.1 Task and Spec Subwindow

Node Definition Window:

The node definition will be organized in a tree structure. Select any node or unselect it at any time by using the left mouse button to click it. Selected nodes will be displayed in inverse video. Use the right mouse button to change the definition of node.

Menu:

- File: Allows read, modify, and save a file.
- Edit:
    - Add: Will add a node as a child of selected node. If no node is selected, it will be added as a root. The user will be prompted for the detailed definition of that node. A task

      node dialog box consist of:
        - Node Name: Write your node name here.
        - Terminal Task: Selects whether you want this be terminal task.
        - Number of inputs: Push this button to bring up another dialog box to fill in input names. Press OK or Cancel when finished.
        - Number of outputs: Push this button to bring up another dialog box to fill in output names. Press OK or Cancel when finished
    - Delete: delete the selected node, all the children of it will be move to the front.
    - Attach: Select a node before using this function, and then click another node (left button). The previously selected node will be attached to the newly selected one.
    - Detach: Detach the selected node.

Drag and Drop:

To add a node from node definition window to editor window, use the middle mouse button, click a node in the node definition window and hold the button down. The node definition window must be on the top of the windows hierarchy. A drag icon will appear. Drag it to the editor window.

Steps:

Task Node:

1. From Main window *Node* menu, select the *Task Node* submenu.

2. From *TaskNode Def* window, edit the definition of any task nodes.

{a) Add new task definition by selecting the *Edit/Add* submenu. When *Add* is selected, enter task names, input specnode names, and output specnode names.

(b) After every task node is added, save the task definition file using either *Save A*s or *Save* under the *File* menu.

(c) *Attach* and *Detach* under the *Edit* menu are used for organizing the task node hierachy. For exarnple, if task A is a superclass of task B; create a relationship A-B by selecting B, clicking the selection mouse button (the left-most button), clicking the *Attach* menu, and choosing A with the selection button.

SpecNode:

Using the SpecNode Definition window is almost identical to that of the TaskNode window, except it does not ask for input and output specs.

Production Editor:

1. Creating the left-hand side of the production. Choose a task from TaskNode window with the middle mouse button. Keep this button pressed and release it at the left-hand side window of the editor window. At this point, all input and output specnodes are connected properly, since these were defined when the task node was created.

2. Choose *Add Alt* as a submenu of the Edit menu. A new window for the right-hand side of the production is just added. Notice that all input and output specs are already there since these are required for the production.

3. Copy and paste by using middle mouse button from *TaskNode Def* or *SpecNode Def* for the alternative. Resize by selecting *Size* under the *Edit* menu and then using the right most button. Move the object by selecting: *Move* under Edit menu using the right most button. Draw edges by clicking the left most button. Click a "from" node and then click a “to" node. If the user tries to connect in an arbitrary fashion, the system will beep and not connect, because the connection is defined in the task node definition.

4. The new productions can be saved using either *Save* or *Save As.*

8. FPGA Synthesis Example

In this section, a synthesis scenario illustrates how the proposed CAD framework can be used. The tools and decompositions employed are intended to be representative, however, not exhaustive.

The circuit being synthesized into is a Field Programmable Gate Array (FPGA) chip is a convolver for a signal processing applications. The primary output is a point multiplication result of input pixels. The objective is to design an FPGA chip from a VHDL behavioral description of the convolver. There are constraints on the number of connections between the FPGA chips and on timing. There is also a constraint on the area of the chip, the most serious limitation of FPGA design.

The functional behavior of the component can be verified via simulation. This simulation and debugging cycle is not part of the synthesis example. Prior to the beginning of synthesis, Cockpit is running with an input file indicating the standard tools and task decompositions available at our site. The primary task, called “FPGA Synthesis”, is initially displayed since this is a goal task. Upon selecting this task, Cockpit tells the user that it can be decomposed into the subtasks “VHDL Compile”, “Place and Route”, and “Bit Generation”. The user asks Cockpit to apply this decomposition and the FPGA Synthesis icon is replaced in the display by the others. The production is shown in Figure 15.

8.1 VHDL Compilation

The transformation of the VHDL behavioral description into the Xilinx netlist file (XNF) and symbol report file generation is the first step of VHDL compilation. When the VHDL Compile is expanded, Cockpit uses the production *elaborate* shown in Figure 16. When this production is applied, *Elaborate* checks the VHDL syntax and transforms the VHDL description into the proper Xilinx netlist file. A portion of the initial VHDL description is shown in Figure 17.

8.2 Placement and Routing

Once the XNF file generation has been completed, the next step is Placement and Routing. In this step, the FPGA logic cells are defined, placed, and routed. Cockpit invokes two tools sequentially: *xnfprep* followed by *ppr* using the production shown in Figure 18. The first tool, *xnfprep,* takes the XNF file as an input and generates the FPGA logic cell definition and PRP report file. Then the second tool, *ppr,* is called to place and route the logic cells onto a FPGA chip.

At this time, ppr cannot finish its placement since the whole logic cannot be fitted into a single Xilinx 4010 FPGA chip. Cockpit detects *ppr's* failure after Cockpit performs the post-evaluation function, e.g., by examining the *ppr* output file. Cockpit tries to backtrack to the Place and Route production to see if there is another alternative for this production, which it tries if available. For this example, however, there is no other alternative. Therefore, Cockpit now moves up to the previous production, VHDL Compile. This production also lacks another alternative, whereupon the whole process fails. Thus, the design should be modified and the process should also be retried.

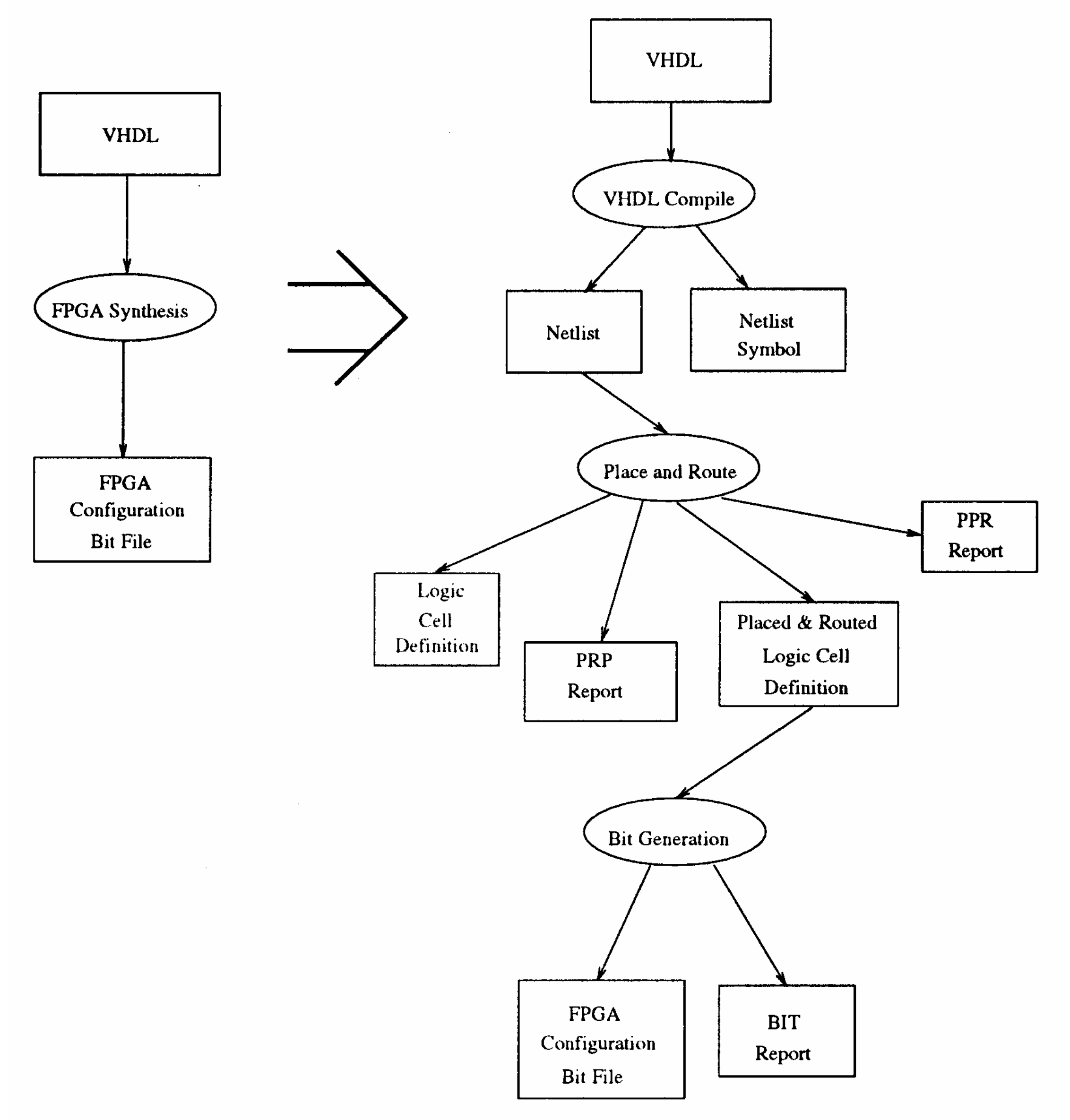


Figure 15. Production of FPGA Synthesis

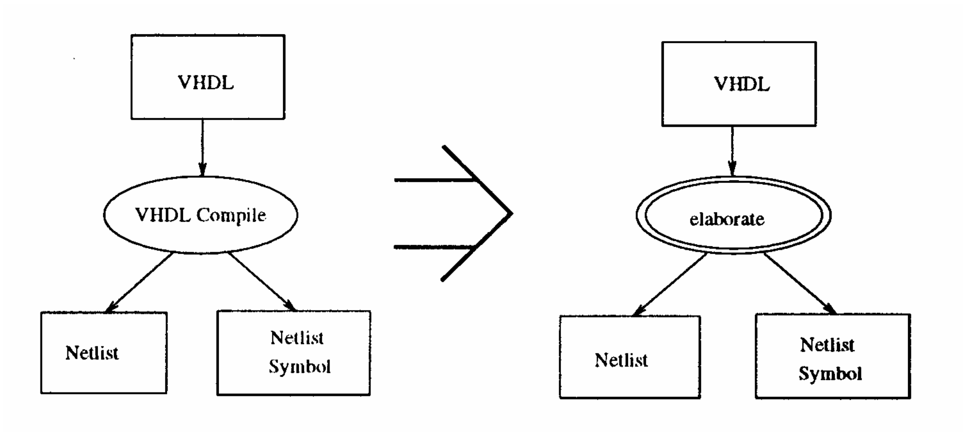


Figure 16. Decomposition of VHDL Compile

```
PROCESS
BEGIN
   WAIT until Xp_Clk'EVENT AND Xp_Clk = '1';
    -- …
   multtemp(31 downto 0) <= itobv(bvtoi(left_in(15 downto 0))
   * bvtoi(left_in(31 downto 16)),32);
   addtemp(31 downto 0) <= itobv(bvtoi(addtemp(31 downto 0))
   + bvtoi(multtemp(31 downto 0)),32);
   right_out{31 downto 0) <= addtempl(31 downto 0);
   right_out(35 downto 32) <= left-in(35 downto 32);
END PROCESS;
```

Figure 17. Initial VHDL Description

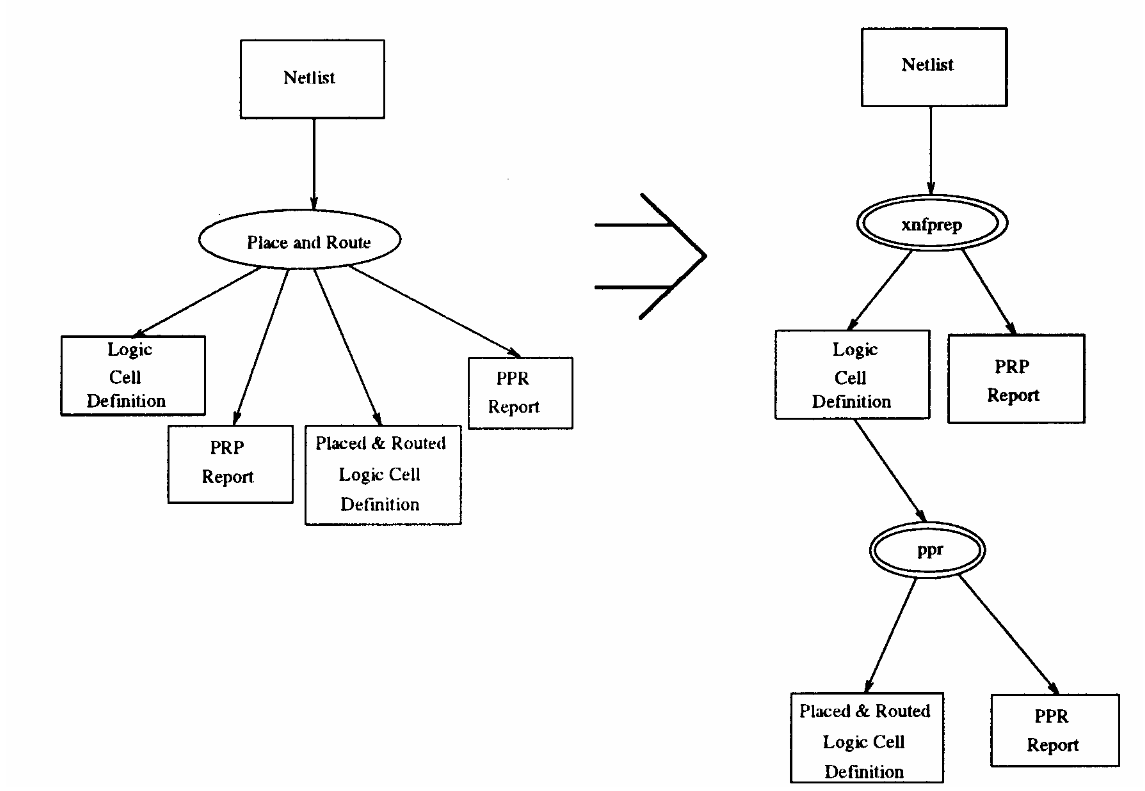


Figure 18. Decomposition of Placement and Route

8.3 Modified Design

The original design is changed so that the new design can fit into a single FPGA chip while maintaining functionality. Since a single 16-bit multiplier takes up a large amount of space, this multiplier is decomposed into four 8-bit multipliers and several adders. A partial description of this decomposition is shown in Figure 19. The new description is used for the same process.

8.4 Synthesis Results

Bit Generation decomposition and all its steps are shown in Figures 20 and 21.

```
PROCESS
BEGIN
   WAIT until Xp_Clk'EVENT AND Xp_Clk = '1';
   -- …
   addtemp1(15 downto 0) <= itobv(bvtoi(left-in(23 downto 16))
            * bvtoi(left_n(7 downto 0)),16);
   addtemp2(23 downto 8) <= itobv(bvtoi(left_in(31 downto 24))
            * bvtoi(left_in(7 downto 0)),16);
   addtemp5(23 downto 8) <= itobv(bvtoi(addtemp1(15 downto 8))
            + bvtoi(addtemp2(23 downto 8)),16);
   addtemp5(7 downto 0) <= addtempl(7 downto 0);
   addtemp3(15 downto 0) <= itobv(bvtoi(left-in(23 downto 16))
            * bvtoi(left-in(15 downto 8)),16);
   addtemp4(23 downto 8) <= itobv(bvtoi(left-in(31 downto 24))
            * bvtoi(left-in(15 downto 8)),16);
   addtemp6(23 downto 8) <= itobv(bvtoi(addtemp3(15 downto 8))
            + bvtoi(addtemp4(23 downto 8)),16);
   addtemp6(7 downto 0) <= addtemp1(7 downto 0);
   -- …
    right-out(35 downto 32) <= left-in(39 downto 32);
END PROCESS;
```

Figure l9. Modified VHDL Description

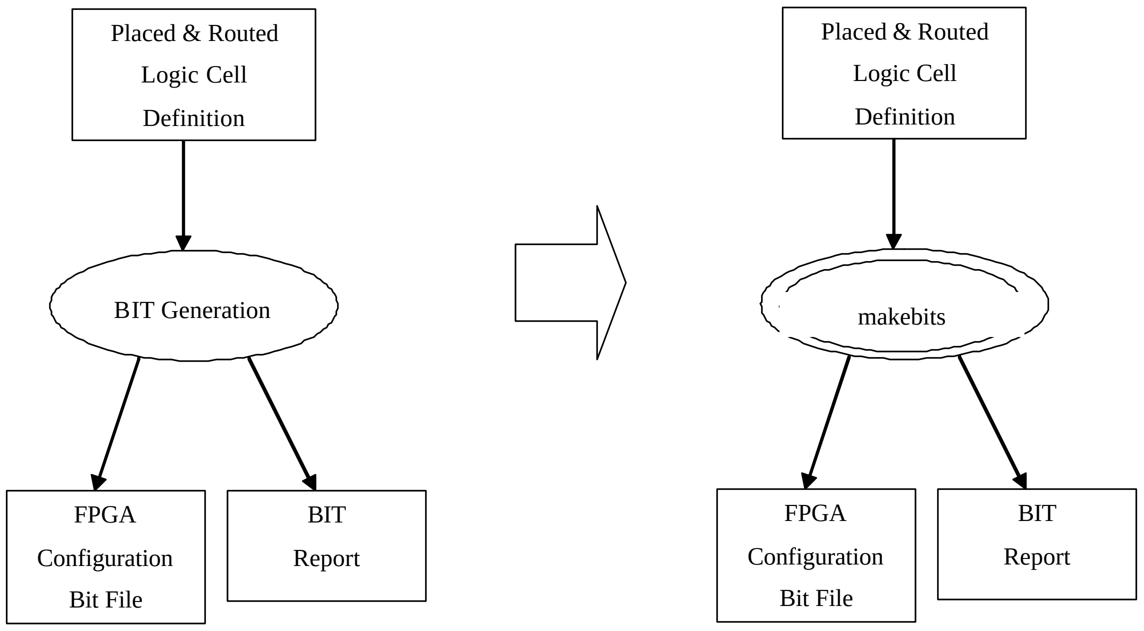

Placed & Routed Logic Cell Definition
BIT Generation
FPGA Configuration Bit File
BIT Report
Placed & Routed Logic Cell Definition
makebits
FPGA Configuration Bit File
BIT Report

Figure 20. Decomposition of Bit Generation

Although the modified design is used for a new synthesis process, the designer can try different constraints and parameters using the original design, For example, one of the constraints subject to change is the *operating condition, w*hich is set to worst case commercial (WCCOM). Examples of parameters are *random seed, placer-effort,* and *router-effort.* As shown in Figure 22, the final design occupies about 98 percent of the available Common Logic Blocks (CLBs) and 68 percent of the available function generators, The maximum speed at which this chip can operate is about 8 MHz, as shown in Figure 22 and 23. In Figure 23, the graph shows that most of the assignments of the nets are done at about a 10 MHz clock rate, although a few of them can operate at a 40 MHz rate. The slowest operations determine the overall rate of the chip's operation speed.

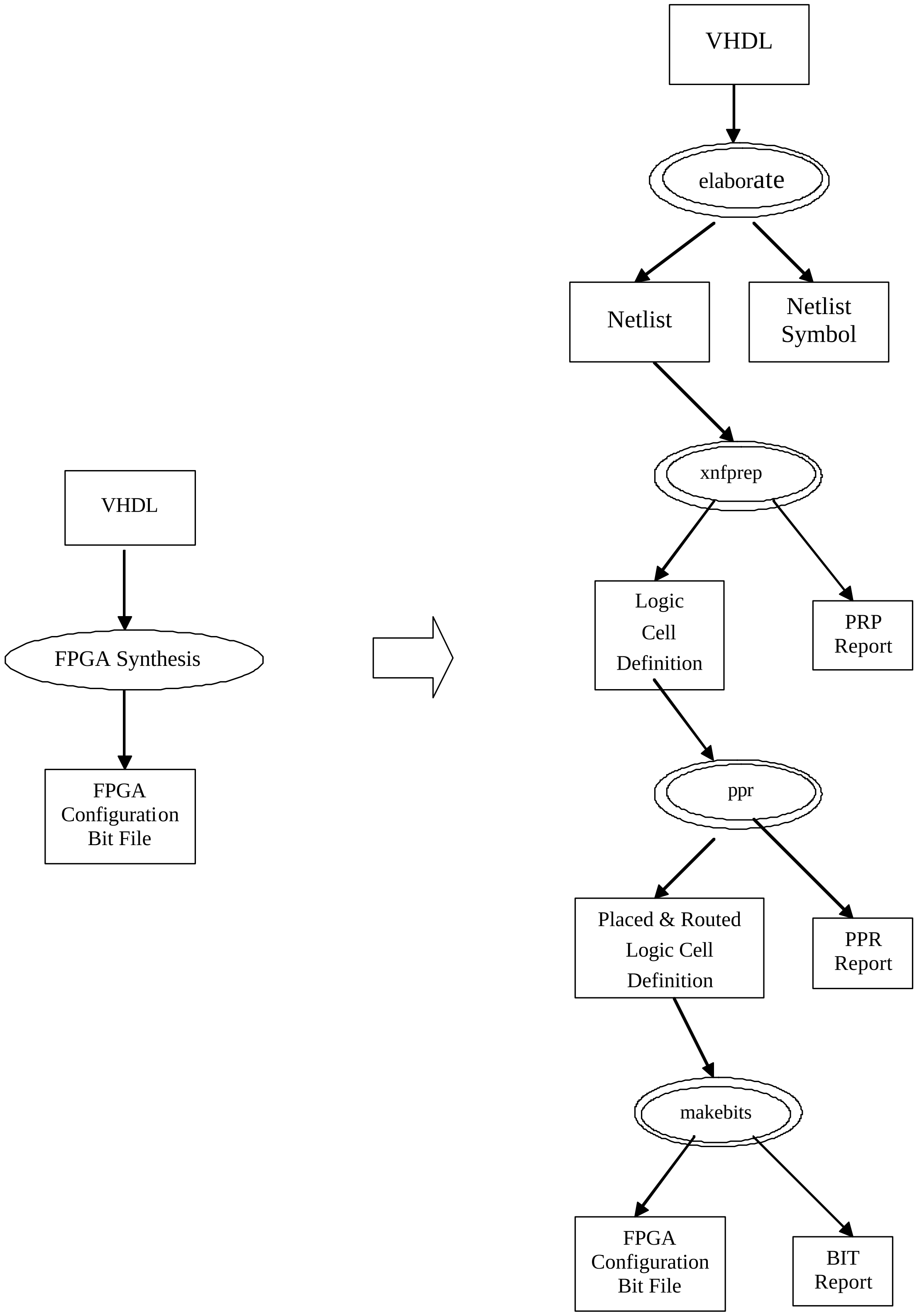


Figure 21. Decomposition of FPGA Synthesis

```
Partitioned Design Utilization Using Part 4010PG191- 6

                                         No. Used   Max Available   % Used
---------------------------------------  ------------  ------------------  ----------
Occupied CLBs                                395          400            98%
 Packed CLBs                                 275          400            68%
---------------------------------------  ------------  ------------------  ----------
 Bonded 1/0 Pins:                             77          160            48%
 F and G Function Generators:                551          800            68%
 H Function Generators:                       58          400            14%
CLB Flip Flops:                               64          800             8%
IOB Input Flip Flops:                          0          160             0%
 IOB Output Flip Flops:                       36          160            22%
Memory Write Controls:                         0          400             0%
3-State Buffers:                               0          880             0%
3-State Half Longlines:                        0           80             0%
Edge Decode Inputs:                            0          240             0%
Edge Decode Half Longlines:                    0           32             0%

            Minimum Clock Period: 125.9 ns
```

Figure 22. PPR and Timing Report Summary

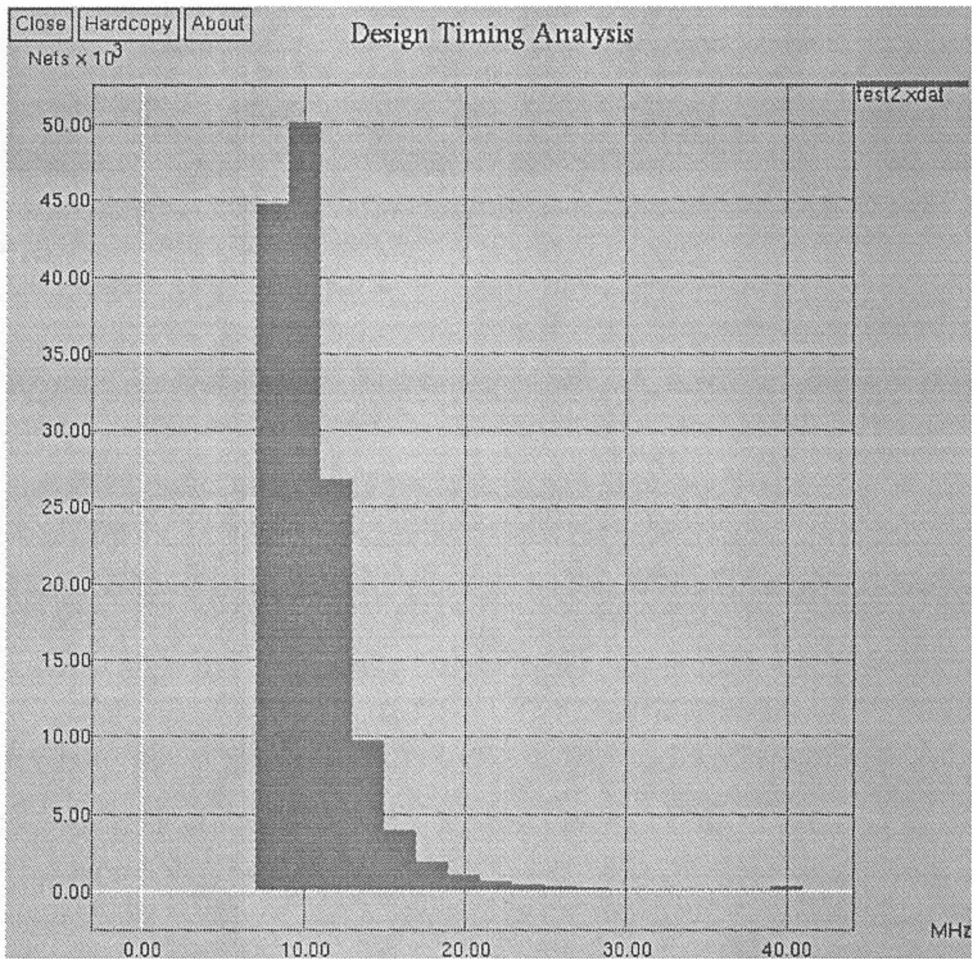


Figure 23. Graphical Timing Result

9. Publications

"Managing Engineering Data for Complex Products" (wlth Reid Baldwin) *Research in Engineering Design,* pp. 215-231, July 1995

"Design Methodology Management: A Formal Approach", {with Reid Baldwin) IiEEE Computer, pp. 54-63, 1995.

"A Path-Oriented Algorithm for the Cell Selection Problem", (with S. Kim) *IEEE Transactions on Computer-Aided Design of Integrated Circuit and Systems,* Vol. 14, No. 3, pp. 296-307, March 1995.

"Object Oriented Modeling for Dynamic Simulation" (with J. Zhou), *Trannsactions on Computer Simulation,* Vol.12, No. l, pp. 1-25.

"Design Methodology Management Using Graph Grammars" (with Reid Baldwin), *Design Automation Conference*, pp. 472-478, 1994.

"Managing a RASSP Design Process", Aeeepted by *Computers in Industry*.

"A VHDL Synthesis Framework", (with Reid Baldwin and Sea Choi), *VHDL Internatlonal Users* Forum Conference, 1994

"Parallel Compiled Mode VHDL Simulation", accepted by *VHDL International User's Conference,* Boston, MA, October, 1995.

"Issues involved in Reuse Library for Design for Test", *Proceedings of AutotestConf'95,* pp. 84-93, Atlanta Georgia, August 1995.

"Using VHDL to a Model Signal Processor", accepted by *VHDL International Users Forum Fall Conference*, November 1994.

"A Path-Oriented Algorithm for the Cell Selection Problems", submitted to *IEEE Transactions on Computer-Aided Design of Integrated Circuit* and *Systems.*

"Methodology of System Design Using VHDL" (with S. Choi), *Proceedings of VHDL Spring 92 Conference,* pp. 11-18.

"A Constraint-Driven Approach to the Configuration Binding in an Object-Oriented VHDL System" (with S. Kim), *Proceedings of Tenth International Symposium on Computer Hardware Design Languages and Their Applications,* pp. 359-374, April 1991.

"The Configuration Management for Version Control in an Object Oriented VHDL Environment" (with S. Kim), *Procedings ICCAD91,* pp. 258-261, November 1991.

10. References

[1] D.G, Fairbairn, "1994 Keynote Address," in *Proceedings of the 31st Design Automation Conference,* pp. xvi-xvii, 1994.

[2] S. Kim, “Configuration Management and Version Data Modeling in VLSI Design Environments”, PhD Thesis, Michigan State University, 1994.

[3] S. Kleinfeldt, M. Guiney, J. K. Miller, and M. Barnes, "Design Methodology Management", *Proceedings of the IEEE, V*ol. 82, pp. 231-250, February 1994.

[4] CF1, "CAD Framework Users, Goals, and Objectives”, *Technical Report Version 0.91*, CAD Framework Initiative, Inc., August 1990.
!
[5] D. S. Harrison, A. R. Newton, R. L. Spickelmier, and T. J. Barnes, "Electronic CAD Frameworks, *ProceedIngs of the IEEE,* Vol. 78, pp. 393-417, February 1990.

[6] R. A. Baldwin, “A Discipline Independent Framework for Engineering Design”, PhD Thesis, MIchIgan State University , 1994.